

Lecture 20 – Type Operators

AAA551: Programming Language Theory

Jihyeok Park



2026 Spring

- Existential Types
 - Syntax and Dynamic Semantics
 - Well-Formedness and Typing Rules
 - Type Equivalence
 - Examples (Counter ADT)
 - Type Erasure
- Subtyping for Existential Types
- Encoding Existential Types via Universal Types

1. Type Operators

Motivation

Kinds

Syntax of System F_ω

Kinding Rules

Type Equivalence

Typing Rules

2. Higher-Kinded Types in Scala

Type Lambdas

Higher-Kinded Type Parameters

3. Subtyping for Type Operators

1. Type Operators

Motivation

Kinds

Syntax of System F_ω

Kinding Rules

Type Equivalence

Typing Rules

2. Higher-Kinded Types in Scala

Type Lambdas

Higher-Kinded Type Parameters

3. Subtyping for Type Operators

So far, every **type abstraction** we have seen $(\forall\alpha.\tau, \exists\alpha.\tau)$ abstracts over a **proper type** α – a type that classifies values.

So far, every **type abstraction** we have seen ($\forall\alpha.\tau$, $\exists\alpha.\tau$) abstracts over a **proper type** α – a type that classifies values.

In Scala, **type constructors** that take types and produce types:

```
List[Int]
```

```
Map[String, Int]
```

```
Option[Bool]
```

So far, every **type abstraction** we have seen ($\forall\alpha.\tau$, $\exists\alpha.\tau$) abstracts over a **proper type** α – a type that classifies values.

In Scala, **type constructors** that take types and produce types:

```
List[Int]
```

```
Map[String, Int]
```

```
Option[Boolean]
```

Here `List`, `Map`, and `Option` are **not** types themselves – they are **functions on types**:

```
List    : Type → Type
```

```
Map     : Type → Type → Type
```

```
Option  : Type → Type
```

So far, every **type abstraction** we have seen ($\forall\alpha.\tau$, $\exists\alpha.\tau$) abstracts over a **proper type** α – a type that classifies values.

In Scala, **type constructors** that take types and produce types:

<code>List[Int]</code>	<code>Map[String, Int]</code>	<code>Option[Bool]</code>
------------------------	-------------------------------	---------------------------

Here `List`, `Map`, and `Option` are **not** types themselves – they are **functions on types**:

`List` : `Type` \rightarrow `Type`

`Map` : `Type` \rightarrow `Type` \rightarrow `Type`

`Option` : `Type` \rightarrow `Type`

Idea: introduce **type-level functions**, with their own notion of application and reduction.

A **type operator** is a function from types to types. For example:

$$\text{Id} \triangleq \lambda\alpha. \alpha$$

$$\text{KNum} \triangleq \lambda\alpha. \text{num}$$

$$\text{Pair} \triangleq \lambda\alpha. \lambda\beta. \forall\gamma. (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \gamma$$

A **type operator** is a function from types to types. For example:

$$\text{Id} \triangleq \lambda\alpha. \alpha$$

$$\text{KNum} \triangleq \lambda\alpha. \text{num}$$

$$\text{Pair} \triangleq \lambda\alpha. \lambda\beta. \forall\gamma. (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \gamma$$

Just as types classify **values**, we need **kinds** – the “types of types” – to classify type operators:

$$\text{Id} :: * \rightarrow * \quad \text{KNum} :: * \rightarrow * \quad \text{Pair} :: * \rightarrow * \rightarrow *$$

A **kind** K classifies type expressions:

$$\begin{array}{l} \text{Kinds } K ::= * \quad (\text{kind of proper types}) \\ \quad \quad | K \rightarrow K \quad (\text{kind of type operators}) \end{array}$$

A **kind** K classifies type expressions:

$$\begin{array}{l} \text{Kinds } K ::= * \quad (\text{kind of proper types}) \\ \quad \quad | K \rightarrow K \quad (\text{kind of type operators}) \end{array}$$

Read $K_1 \rightarrow K_2$ as: “type operator that takes a type of kind K_1 and produces a type of kind K_2 ”.

A **kind** K classifies type expressions:

$$\begin{array}{l} \text{Kinds } K ::= * \quad (\text{kind of proper types}) \\ \quad \quad | K \rightarrow K \quad (\text{kind of type operators}) \end{array}$$

Read $K_1 \rightarrow K_2$ as: “type operator that takes a type of kind K_1 and produces a type of kind K_2 ”.

Examples:

<code>num, bool, num → num</code>	<code>:: *</code>
<code>List, Option, Id, KNum</code>	<code>:: * → *</code>
<code>Pair, Map, Either</code>	<code>:: * → * → *</code>
<code>Container (later in this lecture)</code>	<code>:: (* → *) → *</code>

A **kind** K classifies type expressions:

$$\begin{array}{l} \text{Kinds } K ::= * \quad (\text{kind of proper types}) \\ \quad \quad | K \rightarrow K \quad (\text{kind of type operators}) \end{array}$$

Read $K_1 \rightarrow K_2$ as: “type operator that takes a type of kind K_1 and produces a type of kind K_2 ”.

Examples:

<code>num, bool, num → num</code>	<code>:: *</code>
<code>List, Option, Id, KNum</code>	<code>:: * → *</code>
<code>Pair, Map, Either</code>	<code>:: * → * → *</code>
<code>Container (later in this lecture)</code>	<code>:: (* → *) → *</code>

$\Gamma \vdash e : \tau$ makes sense only when the type τ is a **proper type** of kind $*$.

We replace the well-formedness judgement $\Gamma \vdash \tau$ with the more general **kinding judgement**:

$$\boxed{\Gamma \vdash \tau :: K}$$

read as “in environment Γ , type τ has kind K ”.

We replace the well-formedness judgement $\Gamma \vdash \tau$ with the more general **kinding judgement**:

$$\boxed{\Gamma \vdash \tau :: K}$$

read as “in environment Γ , type τ has kind K ”.

The type environment Γ now binds each type variable to its **kind**:

$$\Gamma ::= \emptyset \mid \Gamma[\alpha :: K] \mid \Gamma[x : \tau]$$

Previously we wrote $\Gamma[\alpha]$ for “ α is a proper type”; this is now shorthand for $\Gamma[\alpha :: *]$.

We extend System F with **type-level abstraction** and **type-level application**:

Expressions $e ::= n \mid x \mid e + e \mid \lambda x:\tau. e \mid e e \mid \Lambda\alpha :: K. e \mid e[\tau]$

Types $\tau ::= \text{num} \mid \tau \rightarrow \tau \mid \alpha \mid \forall\alpha :: K. \tau$
 $\mid \lambda\alpha :: K. \tau \mid \tau[\tau]$

Kinds $K ::= * \mid K \rightarrow K$

We extend System F with **type-level abstraction** and **type-level application**:

Expressions $e ::= n \mid x \mid e + e \mid \lambda x:\tau. e \mid e e \mid \Lambda\alpha :: K. e \mid e[\tau]$

Types $\tau ::= \text{num} \mid \tau \rightarrow \tau \mid \alpha \mid \forall\alpha :: K. \tau$
 $\mid \lambda\alpha :: K. \tau \mid \tau[\tau]$

Kinds $K ::= * \mid K \rightarrow K$

Two new type forms:

- $\lambda\alpha :: K. \tau$ – **type-level abstraction**
- $\tau_1[\tau_2]$ – **type-level application**

We extend System F with **type-level abstraction** and **type-level application**:

Expressions $e ::= n \mid x \mid e + e \mid \lambda x:\tau. e \mid e e \mid \Lambda\alpha :: K. e \mid e[\tau]$

Types $\tau ::= \text{num} \mid \tau \rightarrow \tau \mid \alpha \mid \forall\alpha :: K. \tau$
 $\mid \lambda\alpha :: K. \tau \mid \tau[\tau]$

Kinds $K ::= * \mid K \rightarrow K$

Two new type forms:

- $\lambda\alpha :: K. \tau$ – **type-level abstraction**
- $\tau_1[\tau_2]$ – **type-level application**

The **term-level** syntax and dynamic semantics are essentially unchanged: all new structure lives at the **type level**. F_ω is sometimes described as “STLC on top of System F”.

$$\boxed{\Gamma \vdash \tau :: K}$$

$$\frac{}{\Gamma \vdash \text{num} :: *} \quad \frac{\Gamma \vdash \tau_1 :: * \quad \Gamma \vdash \tau_2 :: *}{\Gamma \vdash \tau_1 \rightarrow \tau_2 :: *} \quad \frac{(\alpha :: K) \in \Gamma}{\Gamma \vdash \alpha :: K}$$

$$\frac{\Gamma[\alpha :: K] \vdash \tau :: *}{\Gamma \vdash \forall \alpha :: K. \tau :: *}$$

$$\frac{\Gamma[\alpha :: K_1] \vdash \tau :: K_2}{\Gamma \vdash \lambda \alpha :: K_1. \tau :: K_1 \rightarrow K_2} \quad \frac{\Gamma \vdash \tau_1 :: K_1 \rightarrow K_2 \quad \Gamma \vdash \tau_2 :: K_1}{\Gamma \vdash \tau_1[\tau_2] :: K_2}$$

$$\boxed{\Gamma \vdash \tau :: K}$$

$$\frac{}{\Gamma \vdash \text{num} :: *} \quad \frac{\Gamma \vdash \tau_1 :: * \quad \Gamma \vdash \tau_2 :: *}{\Gamma \vdash \tau_1 \rightarrow \tau_2 :: *} \quad \frac{(\alpha :: K) \in \Gamma}{\Gamma \vdash \alpha :: K}$$

$$\frac{\Gamma[\alpha :: K] \vdash \tau :: *}{\Gamma \vdash \forall \alpha :: K. \tau :: *}$$

$$\frac{\Gamma[\alpha :: K_1] \vdash \tau :: K_2}{\Gamma \vdash \lambda \alpha :: K_1. \tau :: K_1 \rightarrow K_2} \quad \frac{\Gamma \vdash \tau_1 :: K_1 \rightarrow K_2 \quad \Gamma \vdash \tau_2 :: K_1}{\Gamma \vdash \tau_1[\tau_2] :: K_2}$$

The last two rules are the **STLC typing rules**, lifted to the type level. λ at the type level is the introduction form for $K_1 \rightarrow K_2$, and type application is the elimination form.

With type-level λ and application, two syntactically different types may denote the same thing:

$$(\lambda\alpha :: *. \alpha \rightarrow \alpha)[\text{num}] \quad \text{and} \quad \text{num} \rightarrow \text{num}$$

We extend the equivalence relation with a **type-level β rule**:

$$\boxed{\tau \equiv \tau} \quad \frac{}{(\lambda\alpha :: K. \tau)[\tau'] \equiv \tau[\alpha \mapsto \tau']}$$

With type-level λ and application, two syntactically different types may denote the same thing:

$$(\lambda \alpha :: *. \alpha \rightarrow \alpha)[\text{num}] \quad \text{and} \quad \text{num} \rightarrow \text{num}$$

We extend the equivalence relation with a **type-level β rule**:

$$\boxed{\tau \equiv \tau} \quad \frac{}{(\lambda \alpha :: K. \tau)[\tau'] \equiv \tau[\alpha \mapsto \tau']}$$

Plus the usual congruence rules so \equiv remains a congruence under every type constructor:

$$\frac{\tau_1 \equiv \tau'_1 \quad \tau_2 \equiv \tau'_2}{\tau_1[\tau_2] \equiv \tau'_1[\tau'_2]} \quad \frac{\tau \equiv \tau'}{\lambda \alpha :: K. \tau \equiv \lambda \alpha :: K. \tau'} \quad \dots$$

With type-level λ and application, two syntactically different types may denote the same thing:

$$(\lambda\alpha :: *. \alpha \rightarrow \alpha)[\text{num}] \quad \text{and} \quad \text{num} \rightarrow \text{num}$$

We extend the equivalence relation with a **type-level β rule**:

$$\boxed{\tau \equiv \tau} \quad \frac{}{(\lambda\alpha :: K. \tau)[\tau'] \equiv \tau[\alpha \mapsto \tau']}$$

Plus the usual congruence rules so \equiv remains a congruence under every type constructor:

$$\frac{\tau_1 \equiv \tau'_1 \quad \tau_2 \equiv \tau'_2}{\tau_1[\tau_2] \equiv \tau'_1[\tau'_2]} \quad \frac{\tau \equiv \tau'}{\lambda\alpha :: K. \tau \equiv \lambda\alpha :: K. \tau'} \quad \dots$$

Type-level reduction is **strongly normalizing** (it is just STLC). So \equiv is **decidable**: reduce both sides to normal form and compare syntactically (up to α -equivalence).

Most typing rules are unchanged from System F. The two essential upgrades are:

- replace well-formedness $\Gamma \vdash \tau$ with kinding $\Gamma \vdash \tau :: *$;
- use \equiv (which now includes type-level β) wherever types are compared.

Most typing rules are unchanged from System F. The two essential upgrades are:

- replace well-formedness $\Gamma \vdash \tau$ with kinding $\Gamma \vdash \tau :: *$;
- use \equiv (which now includes type-level β) wherever types are compared.

$$\frac{\Gamma \vdash \tau_1 :: * \quad \Gamma[x : \tau_1] \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1. e : \tau_1 \rightarrow \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_3 \quad \tau_1 \equiv \tau_3}{\Gamma \vdash e_1 e_2 : \tau_2}$$

Most typing rules are unchanged from System F. The two essential upgrades are:

- replace well-formedness $\Gamma \vdash \tau$ with kinding $\Gamma \vdash \tau :: *$;
- use \equiv (which now includes type-level β) wherever types are compared.

$$\frac{\Gamma \vdash \tau_1 :: * \quad \Gamma[x : \tau_1] \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1. e : \tau_1 \rightarrow \tau_2} \quad \frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_3 \quad \tau_1 \equiv \tau_3}{\Gamma \vdash e_1 e_2 : \tau_2}$$

$$\frac{\alpha \notin \text{dom}(\Gamma) \quad \Gamma[\alpha :: K] \vdash e : \tau}{\Gamma \vdash \Lambda \alpha :: K. e : \forall \alpha :: K. \tau} \quad \frac{\Gamma \vdash e : \forall \alpha :: K. \tau' \quad \Gamma \vdash \tau :: K}{\Gamma \vdash e[\tau] : \tau'[\alpha \mapsto \tau]}$$

Most typing rules are unchanged from System F. The two essential upgrades are:

- replace well-formedness $\Gamma \vdash \tau$ with kinding $\Gamma \vdash \tau :: *$;
- use \equiv (which now includes type-level β) wherever types are compared.

$$\frac{\Gamma \vdash \tau_1 :: * \quad \Gamma[x : \tau_1] \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1. e : \tau_1 \rightarrow \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_3 \quad \tau_1 \equiv \tau_3}{\Gamma \vdash e_1 e_2 : \tau_2}$$

$$\frac{\alpha \notin \text{dom}(\Gamma) \quad \Gamma[\alpha :: K] \vdash e : \tau}{\Gamma \vdash \Lambda \alpha :: K. e : \forall \alpha :: K. \tau} \qquad \frac{\Gamma \vdash e : \forall \alpha :: K. \tau' \quad \Gamma \vdash \tau :: K}{\Gamma \vdash e[\tau] : \tau'[\alpha \mapsto \tau]}$$

Notice how the kind annotation on \forall **controls what kind of type** can instantiate it – so we can now write, e.g., $\forall F :: * \rightarrow * \dots$

1. Type Operators

Motivation

Kinds

Syntax of System F_ω

Kinding Rules

Type Equivalence

Typing Rules

2. Higher-Kinded Types in Scala

Type Lambdas

Higher-Kinded Type Parameters

3. Subtyping for Type Operators

Scala 3 supports **type-level abstraction** directly via **type lambdas**:

```
type Id    = [X] =>> X           // identity operator
type KInt  = [X] =>> Int        // constant Int
type Pair  = [X, Y] =>> (X, Y)  // binary product
type Copy  = [X] =>> (X, X)     // copying operator
type Comp[F[_], G[_]] = [X] =>> F[G[X]] // composition
```

The syntax `[X] =>> ...` corresponds directly to $\lambda\alpha :: *. \dots$, and `[X, Y] =>> ...` to a curried type-level λ .

Scala 3 supports **type-level abstraction** directly via **type lambdas**:

```
type Id    = [X] =>> X           // identity operator
type KInt  = [X] =>> Int         // constant Int
type Pair  = [X, Y] =>> (X, Y)   // binary product
type Copy  = [X] =>> (X, X)      // copying operator
type Comp[F[_], G[_]] = [X] =>> F[G[X]] // composition
```

The syntax `[X] =>> ...` corresponds directly to $\lambda\alpha :: *. \dots$, and `[X, Y] =>> ...` to a curried type-level λ .

```
val n: Id[Int]          = 42
val m: KInt[Bool]      = 42
val p: Pair[Int, String] = (1, "two")
// Comp[Copy, Copy][Int] = Copy[Copy[Int]] = ((Int, Int), (Int, Int))
val c: Comp[Copy, Copy][Int] = ((1, 2), (3, 4))
```

The kind $(* \rightarrow *) \rightarrow *$ corresponds to a **higher-kinded type parameter**, written $F[_]$ in Scala. For example, a generic Container abstraction:

```
trait Container[F[_]]:                                     // F :: * -> *
  def empty[A]: F[A]
  def size[A](fa: F[A]): Int
```

Here $\text{Container}[F[_]]$ is parameterized by a **type operator** $F :: * \rightarrow *$.

The kind $(* \rightarrow *) \rightarrow *$ corresponds to a **higher-kinded type parameter**, written $F[_]$ in Scala. For example, a generic Container abstraction:

```
trait Container[F[_]]:                                     // F :: * -> *
  def empty[A]: F[A]
  def size[A](fa: F[A]): Int
```

Here $\text{Container}[F[_]]$ is parameterized by a **type operator** $F :: * \rightarrow *$.

```
given Container[List] with
  def empty[A]: List[A] = Nil
  def size[A](xs: List[A]): Int = xs.length

given Container[Option] with
  def empty[A]: Option[A] = None
  def size[A](o: Option[A]): Int = if o.isEmpty then 0 else 1
```

We use a `Container` instance through Scala 3's **context parameter** (`using` clause):

```
def isEmpty[F[_], A](fa: F[A])(using C: Container[F]): Boolean =
  C.size(fa) == 0

isEmpty(List(1, 2, 3))           // false (Container[List] used)
isEmpty(Option.empty[Int])      // true  (Container[Option] used)
isEmpty(Option(true))          // false (Container[Option] used)
```

The compiler **infers** `F` from the argument (e.g., `F = List`) and then **synthesizes** the matching `Container[F]` instance.

We use a Container instance through Scala 3's **context parameter** (**using** clause):

```
def isEmpty[F[_], A](fa: F[A])(using C: Container[F]): Boolean =
  C.size(fa) == 0

isEmpty(List(1, 2, 3))           // false (Container[List] used)
isEmpty(Option.empty[Int])      // true  (Container[Option] used)
isEmpty(Option(true))          // false (Container[Option] used)
```

The compiler **infers** F from the argument (e.g., $F = \text{List}$) and then **synthesizes** the matching $\text{Container}[F]$ instance.

Equivalent shorthand using a **context bound** $F[_]: \text{Container}$:

```
def isEmpty[F[_]: Container, A](fa: F[A]): Boolean =
  summon[Container[F]].size(fa) == 0
```

What if we want a `Container` instance for `Either[String, _]`? `Either` has kind $* \rightarrow * \rightarrow *$, but `Container` wants something of kind $* \rightarrow *$.

What if we want a Container instance for `Either[String, _]`? `Either` has kind $* \rightarrow * \rightarrow *$, but `Container` wants something of kind $* \rightarrow *$.

Solution: **partially apply** `Either` via a type lambda to get something of kind $* \rightarrow *$:

```
given Container[[X] =>> Either[String, X]] with
  def empty[A]: Either[String, A] = Left("empty")
  def size[A](e: Either[String, A]): Int = if e.isRight then 1 else 0
```

What if we want a `Container` instance for `Either[String, _]`? `Either` has kind $* \rightarrow * \rightarrow *$, but `Container` wants something of kind $* \rightarrow *$.

Solution: **partially apply** `Either` via a type lambda to get something of kind $* \rightarrow *$:

```
given Container[[X] =>> Either[String, X]] with
  def empty[A]: Either[String, A] = Left("empty")
  def size[A](e: Either[String, A]): Int = if e.isRight then 1 else 0
```

In the formal system, this is exactly:

$$\text{Container}[\lambda\alpha :: *. \text{Either}[\text{String}][\alpha]]$$

– a type operator passed as an argument to another type operator.

1. Type Operators

Motivation

Kinds

Syntax of System F_ω

Kinding Rules

Type Equivalence

Typing Rules

2. Higher-Kinded Types in Scala

Type Lambdas

Higher-Kinded Type Parameters

3. Subtyping for Type Operators

Combining F_ω (type operators) with $F_{<}$: (bounded quantification) gives $F_{<}^\omega$. The subtype relation must be lifted to **operators of higher kinds**.

Combining F_ω (type operators) with $F_{<}$: (bounded quantification) gives $F_{<}^\omega$. The subtype relation must be lifted to **operators of higher kinds**.

Subtyping is **kind-indexed**: $\Gamma \vdash \tau_1 <: \tau_2 :: K$ means τ_1 and τ_2 both have kind K , and τ_1 is a subtype of τ_2 .

Combining F_ω (type operators) with $F_{<}$: (bounded quantification) gives $F_{<}^\omega$. The subtype relation must be lifted to **operators of higher kinds**.

Subtyping is **kind-indexed**: $\Gamma \vdash \tau_1 <: \tau_2 :: K$ means τ_1 and τ_2 both have kind K , and τ_1 is a subtype of τ_2 .

At an arrow kind, operators are compared **pointwise on the body**:

$$\frac{\Gamma[\alpha :: K_1] \vdash \tau_1 <: \tau_2 :: K_2}{\Gamma \vdash \lambda\alpha :: K_1.\tau_1 <: \lambda\alpha :: K_1.\tau_2 :: K_1 \rightarrow K_2}$$

Combining F_ω (type operators) with $F_{<}$: (bounded quantification) gives $F_{<}^\omega$. The subtype relation must be lifted to **operators of higher kinds**.

Subtyping is **kind-indexed**: $\Gamma \vdash \tau_1 <: \tau_2 :: K$ means τ_1 and τ_2 both have kind K , and τ_1 is a subtype of τ_2 .

At an arrow kind, operators are compared **pointwise on the body**:

$$\frac{\Gamma[\alpha :: K_1] \vdash \tau_1 <: \tau_2 :: K_2}{\Gamma \vdash \lambda\alpha :: K_1.\tau_1 <: \lambda\alpha :: K_1.\tau_2 :: K_1 \rightarrow K_2}$$

Subtyping is also compatible with type-level application:

$$\frac{\Gamma \vdash \tau_1 <: \tau_2 :: K_1 \rightarrow K_2 \quad \Gamma \vdash \tau :: K_1}{\Gamma \vdash \tau_1[\tau] <: \tau_2[\tau] :: K_2}$$

The argument type τ should be the same (invariant) on both sides because we don't know what variance the operator has in its parameter.

For a unary type operator F , we say F is

- **covariant** if $\tau_1 <: \tau_2 \implies F[\tau_1] <: F[\tau_2]$
- **contravariant** if $\tau_1 <: \tau_2 \implies F[\tau_2] <: F[\tau_1]$
- **invariant** otherwise

For a unary type operator F , we say F is

- **covariant** if $\tau_1 <: \tau_2 \implies F[\tau_1] <: F[\tau_2]$
- **contravariant** if $\tau_1 <: \tau_2 \implies F[\tau_2] <: F[\tau_1]$
- **invariant** otherwise

These are properties of how the operator's **body** uses its parameter:

$\lambda\alpha :: *.List[\alpha]$ (covariant) $\lambda\alpha :: *. \alpha \rightarrow \text{num}$ (contravariant)

For a unary type operator F , we say F is

- **covariant** if $\tau_1 <: \tau_2 \implies F[\tau_1] <: F[\tau_2]$
- **contravariant** if $\tau_1 <: \tau_2 \implies F[\tau_2] <: F[\tau_1]$
- **invariant** otherwise

These are properties of how the operator's **body** uses its parameter:

$\lambda\alpha :: *.List[\alpha]$ (covariant) $\lambda\alpha :: *. \alpha \rightarrow \text{num}$ (contravariant)

Scala turns variance into an **annotation** on the operator's parameter:

```
trait List[+A]           // covariant: List[Dog] <: List[Animal]
trait Function1[-A, +B] // contravariant in A, covariant in B
trait Array[A]          // invariant
```

For a unary type operator F , we say F is

- **covariant** if $\tau_1 <: \tau_2 \implies F[\tau_1] <: F[\tau_2]$
- **contravariant** if $\tau_1 <: \tau_2 \implies F[\tau_2] <: F[\tau_1]$
- **invariant** otherwise

These are properties of how the operator's **body** uses its parameter:

$\lambda \alpha :: *. \text{List}[\alpha]$ (covariant) $\lambda \alpha :: *. \alpha \rightarrow \text{num}$ (contravariant)

Scala turns variance into an **annotation** on the operator's parameter:

```
trait List[+A]           // covariant: List[Dog] <: List[Animal]
trait Function1[-A, +B] // contravariant in A, covariant in B
trait Array[A]          // invariant
```

In $F_{<}^\omega$, variance is **derivable** from how α appears in the body; Scala's annotations are kind-level constraints the compiler checks against the body's structure.

- Curry-Howard Isomorphism

Jihyeok Park
jihyeok_park@korea.ac.kr
<https://plrg.korea.ac.kr>